Suggested Solutions to Midterm Problems

1. Reprove the following theorem which we have proven (mostly) in class. This time you must apply the *reversed induction* principle, or a variant of it, in some part of the proof.

   > There exist Gray codes of length $\lceil \log_2 k \rceil$ for any positive integer $k \geq 2$. The Gray codes for the *even* values of k are *closed*, and the Gray codes for *odd* values of k are *open*.

   Solution. To facilitate the proof, we introduce the notion of a symmetric Gray code. A Gray code $s_1, s_2, \ldots, s_{2n-1}, s_{2n}$ for $2n$ objects is said to be *symmetric* if the n pairs s_n and s_{n+1}, s_{n-1} and s_{n+2}, $\ldots$, s_2 and s_{2n-1}, and s_1 and s_{2n} all differ by exactly 1 bit. The motivation behind this is that, given a symmetric Gray code $s_1, s_2, \ldots, s_{n-1}, s_n, s_{n+1}, s_{n+2}, \ldots, s_{2n-1}, s_{2n}$ for $2n$ ($n \geq 1$) objects, we can easily obtain a Gray code for $2n-2$ objects by deleting s_n and s_{n+1}. The resulting Gray code $s_1, s_2, \ldots, s_{n-1}, s_{n+2}, \ldots, s_{2n-1}, s_{2n}$ is again symmetric.

   We then strengthen the theorem by adding the requirement of symmetry to the cases of even values of k, as follows:

   > There exist Gray codes of length $\lceil \log_2 k \rceil$ for any positive integer $k \geq 2$. The Gray codes for the *even* values of k are *closed* and *symmetric*, and the Gray codes for *odd* values of k are *open*.

   The rest of the proof is left as an exercise. □

2. Consider the following two-player game: given a positive integer N, player A and player B take turns counting to N. In his turn, a player may advance the count by 1 or 2. For example, player A may start by saying "1, 2", player B follows by saying "3", player A follows by saying "4", etc. The player who eventually has to say the number N loses the game.

   A game is *determined* if one of the two players always has a way to win the game. Prove that the counting game as described is determined for any positive integer N; the winner may differ for different given integers. You must use induction in your proof. (Hint: think about the remainder of the number N divided by 3.)

   Solution. We first prove the following claim:

   > When $N = 3k + 1$ for some $k \geq 0$, player B can always win the game.

   The proof is by induction on k.

   Base case ($k = 0$, i.e., $N = 1$): player A has no other choice but say 1 and hence player B wins.

   Inductive step ($k \geq 1$, i.e., $N = 3k + 1 \geq 4$): player A starts either by "1" or "1, 2". In both cases, player B can always count to 3. At this point we have the situation analogous to that the two players are to play a game with $N = 3(k-1) + 1$, in which player B can always win from the induction hypothesis.

We next show that, when $N = 3k + 2$ or $N = 3(k + 1)$ for some $k \geq 0$, player A can always win the game. In the case when $N = 3k + 2$, player A starts by saying "1", while in the case when $N = 3(k + 1)$, he starts by "1, 2". After player A's first turn, we have the situation analogous to that player B is to start a game with $N = 3k + 1$, playing the role of player A (to start first in the remaining game). From the preceding claim, player A (playing the role of player B in the remaining game) will win the game.

Now we see that, for every positive integer N, there is always a player that can win the counting game and hence the game is determined. □

3. For each of the following pairs of functions, determine whether $f(n) = O(g(n))$ and/or $f(n) = \Omega(g(n))$. Justify your answers.

	$f(n)$	$g(n)$
(a)	$\frac{n^2}{\log n}$	$n(\log n)^2$
(b)	$n^3 2^n$	3^n

Solution. (Jen-Feng Shih)

(a) $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{n(\log n)^2}{\frac{n^2}{\log n}} = \lim_{n\to\infty} \frac{(\log n)^3}{n} = 0$. Hence, $g(n) = o(f(n))$. It follows that $f(n) = \Omega(g(n))$, but $f(n) \neq O(g(n))$.

Alternatively, $\forall n \geq 2$, $f(n) = \frac{n^2}{\log n} \geq n(\log n)^2 = g(n)$, which shows that $f(n) = \Omega(g(n))$.

(b) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^3 2^n}{3^n} = \lim_{n\to\infty} n^3(\frac{2}{3})^n = 0$. So, $f(n) = o(g(n))$. It follows that $f(n) = O(g(n))$ (and $g(n) = \Omega(f(n))$), but $f(n) \neq \Omega(g(n))$.

Alternatively, $\forall n \geq 7$, $f(n) = n^3 2^n \leq 7^3 \times 3^n = 343 \times g(n)$, which shows that $f(n) = O(g(n))$. The number 7 was decided by considering the largest number n such that $(\frac{n+1}{n})^3 \geq \frac{3}{2}$, after which point the increase via n^3 can never catch up with the increase via $(\frac{3}{2})^n$. □

4. The Knapsack Problem is defined as follows: Given a set S of n items, where the i-th item has an integer size $S[i]$, and an integer K, find a subset of the items whose sizes sum to exactly K or determine that no such subset exists.

Now consider a variant where we want the subset to be as large as possible (i.e., to be with as many items as possible). How will you adapt the algorithm (see the Appendix) that we have studied in class? Your algorithm should collect at the end the items in one of the best solutions if they exist. Please present your algorithm in an adequate pseudo code and make assumptions wherever necessary (you may reuse the code for the original Knapsack Problem). Give an analysis of its time complexity. The more efficient your algorithm is, the more points you will get for this problem.

Solution. (Yi-Wen Chang)

To find the largest possible subset of items, we modify the *Knapsack* algorithm in the Appendix to obtain *Knapsack_ForMaxSubset*, as shown below. Each element $P[i, k]$ in the result array P now contains a new integer variable *size* which memorizes the size of the current largest subset for k.

Algorithm *Knapsack_ForMaxSubset*(S, K)**;**
begin

```
    P[0,0].exist := true;
    P[0,0].size := 0;
    for k := 1 to K do
        P[0,k].exist := false;
        P[0,k].size := 0;
    for i := 1 to n do
        for k := 0 to K do
            P[i,k].exist := false;
            P[i,k].size := 0;
            if k − S[i] ≥ 0 and P[i−1, k−S[i]].exist then
                if P[i−1,k].exist and P[i−1,k].size ≥ P[i−1, k−S[i]].size + 1 then
                    P[i,k].exist := true;
                    P[i,k].belong := false;
                    P[i,k].size := P[i−1,k].size;
                else
                    P[i,k].exist := true;
                    P[i,k].belong := true
                    P[i,k].size := P[i−1, k−S[i]].size + 1;
            else if P[i−1,k].exist then
                P[i,k].exist := true;
                P[i,k].belong := false;
                P[i,k].size := P[i−1,k].size;
    if ¬P[n,K].exist then
      print "no solution"
    else i := n;
         k := K;
         while k > 0 do
            if P[i,k].belong then
                print i;
                k := k − S[i];
            i := i − 1;
end
```

The complexity remains the same, which is $O(nK)$. When a solution (a subset of items whose sizes sum to exactly K) exists, the printed result will be the largest among such subsets. □

5. Let x_1, x_2, ..., x_n be a set of integers, and let $S = \sum_{i=1}^{n} x_i$. Design an algorithm to partition the set into two subsets of equal sum, or determine that it is impossible to do so. When the partitioning is possible, your algorithm should also give the two subsets of integers. The algorithm should run in time $O(nS)$.

Solution. (Jen-Feng Shih)

```
Algorithm Partition_into_Two_Subsets(x);
begin
    sum := Σ_{i=1}^{n} x_i;
    if sum is odd then print "no solution."
    else
        K := sum/2;
```

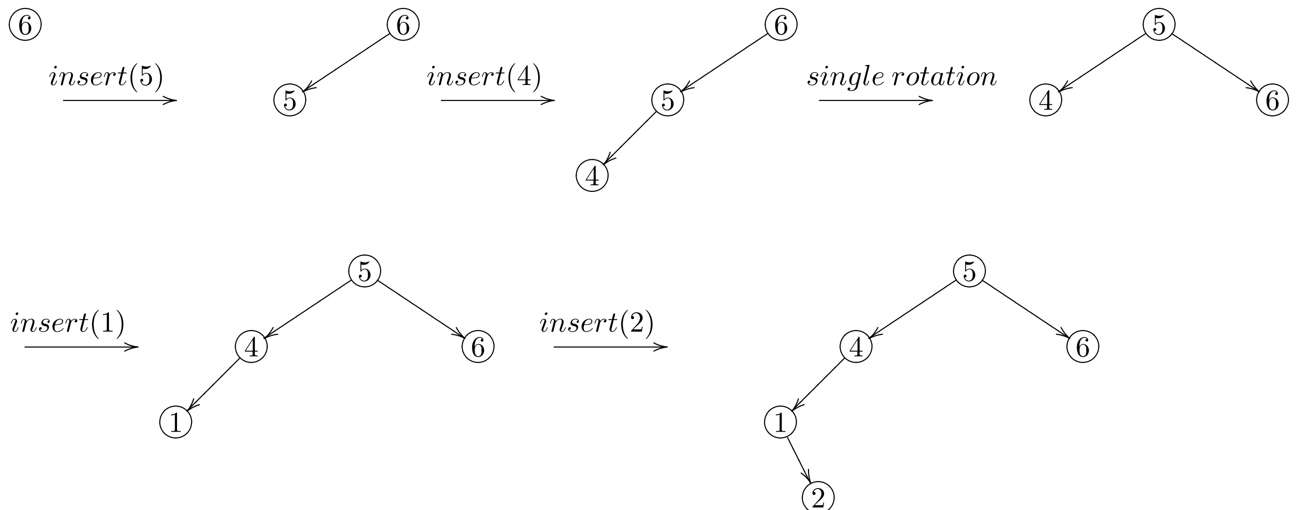
$Knapsack(x, K)$;
if $P[n, K].exist := false$ **then**
  print "no solution."
else
  $l := 1$;
  $m := 1$;
  for $i := n$ **to** 1 **do**
    if $P[i, k].belong := true$ **then**
      $set1[l] := x[i]$;
      $l := l + 1$;
      $k := k - x[i]$;
    else
      $set2[m] := x[i]$;
      $m := m + 1$;
  print "set1:"
  for $i := 1$ **to** $l - 1$ **do**
    print $set1[i]$;
  print "set2:"
  for $i := 1$ **to** $m - 1$ **do**
    print $set2[i]$;
end

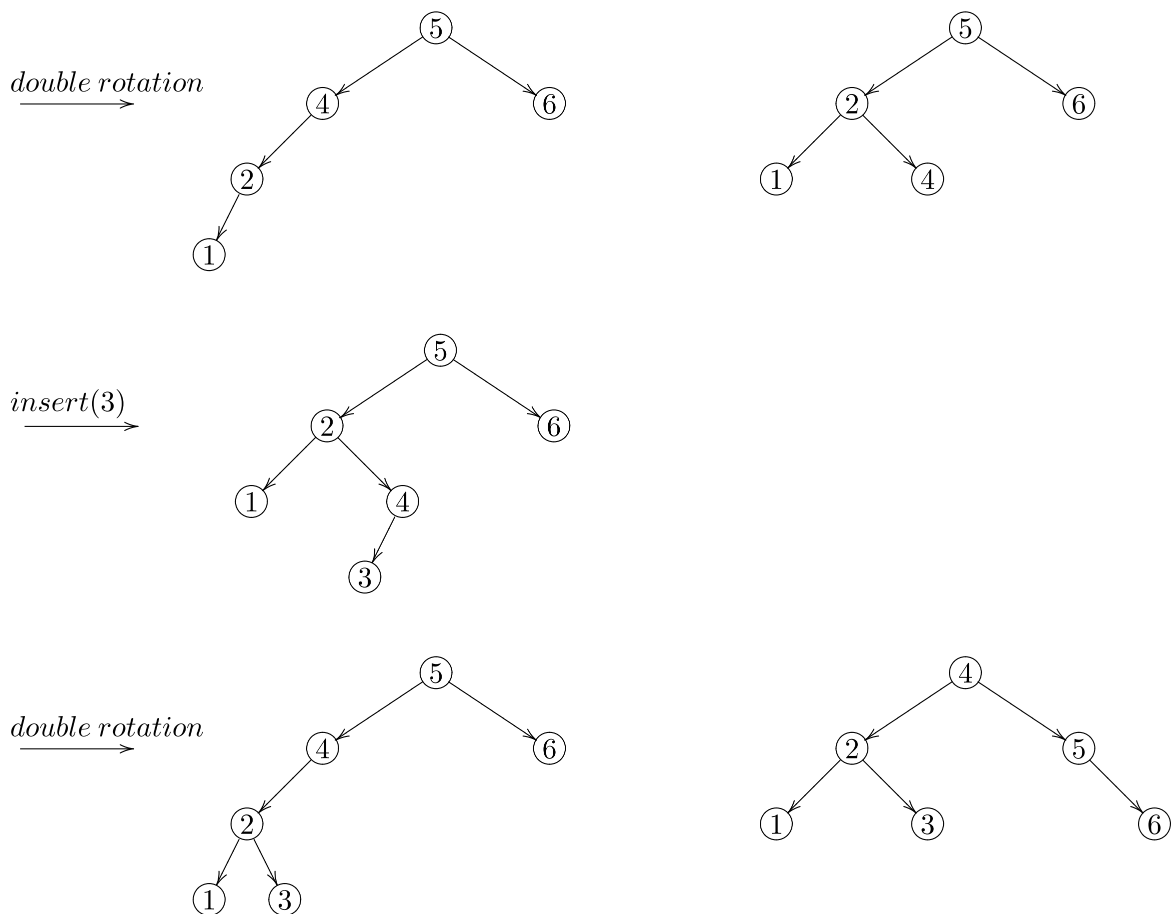
The complexity remains the same as in the Knapsack Problem, which is $O(nK) = O(nS)$.

□

6. Show all intermediate and the final AVL trees formed by inserting the numbers 6, 5, 4, 1, 2, and 3 (in this order) into an empty tree. Please use the following ordering convention: the key of an internal node is larger than that of its left child and smaller than that of its right child. If re-balancing operations are performed, please also show the tree before re-balancing and indicate what type of rotation is used in the re-balancing.

Solution. (Yi-Wen Chang)





□

7. Let $G(h)$ denote the least possible number of nodes contained in an AVL tree of height h. Let us assume that the empty tree has height -1 and a single-node tree has height 0. Please give a recurrence relation that characterizes (fully defines) G. Based on the recurrence relation, prove that the height of an AVL tree of n nodes is $O(\log n)$.

   Solution. The recurrence relation can be defined as follows:

$$\begin{cases} G(-1) & = 0 \\ G(0) & = 1 \\ G(h) & = G(h-1) + G(h-2) + 1, \quad h \geq 1 \end{cases}$$

   A precise solution to $G(h)$ may be derived by establishing the relation $G(h) = F(h+3) - 1$, where $F(n)$ is the n-th Fibonacci number (as defined in Chapter 3.5 of Manber's book) for which we already know the closed form; the proof is in fact quite simple by induction. However, we will prove directly a lower bound for $G(h)$, namely $\Omega((\frac{3}{2})^h)$, which is good enough to show its exponential growth. The proof is by induction on h, showing that $G(h) \geq \frac{2}{3}(\frac{3}{2})^h$, for $h \geq 0$.

   Base case ($h = 0$ or $h = 1$): When $h = 0$, $\frac{2}{3}(\frac{3}{2})^0 = \frac{2}{3} \leq 1 = G(0)$. When $h = 1$, $\frac{2}{3}(\frac{3}{2})^1 = 1 \leq 2 = G(1)$.

   Inductive step ($h > 1$): $G(h) = G(h-1) + G(h-2) + 1$, which from the induction hypothesis $\geq \frac{2}{3}(\frac{3}{2})^{h-1} + \frac{2}{3}(\frac{3}{2})^{h-2} + 1 \geq (1 + \frac{2}{3})(\frac{3}{2})^{h-2} = (1 + \frac{2}{3})(\frac{3}{2})^{-2}(\frac{3}{2})^h = \frac{20}{27}(\frac{3}{2})^h \geq \frac{2}{3}(\frac{3}{2})^h$.

Therefore, for an AVL tree of size n, its height h must be such that $\frac{2}{3}(\frac{3}{2})^h \leq G(h) \leq n$. It follows that $h \leq \frac{1}{\log 1.5} \log n + 1$ (base 2 logarithm), implying $h = O(\log n)$. □

8. The *Partition* procedure for the Quicksort algorithm discussed in class is as follows, where *Middle* is a global variable.

Partition $(X, \mathit{Left}, \mathit{Right})$;
begin
  $\mathit{pivot} := X[\mathit{left}]$;
  $L := \mathit{Left}$; $R := \mathit{Right}$;
  while $L < R$ **do**
    while $X[L] \leq \mathit{pivot}$ and $L \leq \mathit{Right}$ **do** $L := L + 1$;
    while $X[R] > \mathit{pivot}$ and $R \geq \mathit{Left}$ **do** $R := R - 1$;
    if $L < R$ **then** $\mathit{swap}(X[L], X[R])$;
  $\mathit{Middle} := R$;
  $\mathit{swap}(X[\mathit{Left}], X[\mathit{Middle}])$
end

Find an adequate loop invariant for the main while loop, which is sufficient to show that after the execution of the last two assignment statements the array is properly partitioned by $X[\mathit{Middle}]$. Please express the loop invariant as precisely as possible, using mathematical notation.

Solution. The algorithm assumes that $\mathit{Left} < \mathit{Right}$. This condition holds throughout the algorithm and we will keep it implicit. A suitable loop invariant for the main while loop is as follows:

$$\begin{array}{ll} & \mathit{pivot} = X[\mathit{Left}] \\ \wedge & \forall i(\mathit{Left} \leq i < L \rightarrow X[i] \leq \mathit{pivot}) \\ \wedge & \forall j(R < j \leq \mathit{Right} \rightarrow \mathit{pivot} < X[j]) \\ \wedge & \mathit{Left} \leq L \leq \mathit{Right} \\ \wedge & \mathit{Left} \leq R \leq \mathit{Right} \\ \wedge & (L \not< R) \rightarrow (L - 1 = R) \end{array}$$

This loop invariant is maintained before and after every iteration of the loop. Note that the inequalities $i < L$ and $R < j$ in the second and third conjuncts are strict. This is so because when the condition $L < R$ does not hold, the statement $\mathit{swap}(X[L], X[R])$ will not be performed. After the while loop terminates with $L \not< R$ and the following two statements are executed, we can conclude:

$$\begin{array}{ll} & \mathit{pivot} = X[\mathit{Middle}] \\ \wedge & \forall i(\mathit{Left} \leq i \leq \mathit{Middle} \rightarrow X[i] \leq \mathit{pivot}) \\ \wedge & \forall j(\mathit{Middle} < j \leq \mathit{Right} \rightarrow \mathit{pivot} < X[j]) \end{array}$$

which is the (post-)condition desired of the Partition algorithm, indicating that the algorithm is indeed correct. □

9. Consider rearranging the following array into a max heap using the *bottom-up* approach.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	7	1	5	2	14	6	11	8	4	10	15	13	12	9

Please show the result (i.e., the contents of the array) after a new element is added to the current collection of heaps (at the bottom) until the entire array has become a heap.

Solution. (Yi-Wen Chang)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	7	1	5	2	14	6	11	8	4	10	15	13	12	9
3	7	1	5	2	14	12	11	8	4	10	15	13	6	9
3	7	1	5	2	15	12	11	8	4	10	14	13	6	9
3	7	1	5	10	15	12	11	8	4	2	14	13	6	9
3	7	1	11	10	15	12	5	8	4	2	14	13	6	9
3	7	15	11	10	14	12	5	8	4	2	1	13	6	9
3	11	15	8	10	14	12	5	7	4	2	1	13	6	9
15	11	14	8	10	13	12	5	7	4	2	1	3	6	9

□

10. Prove that the sum of the heights of all nodes in a complete binary tree with n nodes is at most $n-1$. You may assume it is known that the sum of the heights of all nodes in a *full* binary tree of height h is $2^{h+1}-h-2$. (Note: a single-node tree has height 0.)

Solution. Let $G(n)$ denote the sum of the heights of all nodes in a complete binary tree with n nodes. For a full binary tree (a special case of complete binary trees) with $n = 2^{h+1}-1$ nodes where h is the height of the tree, we already know that $G(n) = 2^{h+1}-(h+2) = n-(h+1) \leq n-1$. With this as a basis, we prove the general case of arbitrary complete binary trees by induction on the number n (≥ 1) of nodes.

Base case ($n = 1$ or $n = 2$): When $n = 1$, the tree is the smallest full binary tree with one single node whose height is 0. So, $G(n) = 0 \leq 1-1 = n-1$. When $n = 2$, the tree has one additional node as the left child of the root. The height of the root is 1, while that of its left child is 0. So, $G(n) = 1 \leq 2-1 = n-1$.

Inductive step ($n > 2$): If n happens to be equal to $2^{h+1}-1$ for some $h \geq 1$, i.e., the tree is full, then we are done; note that this covers the case of $n = 3 = 2^{1+1}-1$. Otherwise, suppose $2^{h+1}-1 < n < 2^{h+2}-1$ ($h \geq 1$), i.e., the tree is a "proper" complete binary tree with height $h+1 \geq 2$. We observe that at least one of the two subtrees of the root is full, while the other is complete (possibly full). There are three cases to consider:

Case 1: The left subtree is full with n_1 nodes and the right one is complete but not full with n_2 nodes (such that $n_1+n_2+1 = n$). In this case, both subtrees much be of height h and $n_1 = 2^{h+1}-1$. ¿From the special case of full binary trees and the induction hypothesis, $G(n_1) = 2^{h+1}-(h+2) = n_1-(h+1)$ and $G(n_2) \leq n_2-1$. $G(n) = G(n_1)+G(n_2)+(h+1) \leq (n_1-(h+1))+(n_2-1)+(h+1) = (n_1+n_2+1)-2 \leq n-1$.

Case 2: The left subtree is full with n_1 nodes and the right one is also full with n_2 nodes. In this case, the left subtree much be of height h and $n_1 = 2^{h+1}-1$, while the right subtree much be of height $h-1$ and $n_2 = 2^h-1$. ¿From the special case of full binary trees, $G(n_1) = 2^{h+1}-(h+2) = n_1-(h+1)$ and $G(n_2) = 2^h-(h+1) = n_2-h$. $G(n) = G(n_1)+G(n_2)+(h+1) \leq (n_1-(h+1))+(n_2-h)+(h+1) = (n_1+n_2+1)-(h+1) \leq n-1$.

Case 3: The left subtree is complete but not full with n_1 nodes and the right one is full with n_2 nodes. In this case, the left subtree much be of height h, while the right subtree much be of height $h-1$ and $n_2 = 2^h-1$. ¿From the induction hypothesis and the special case of full binary trees, $G(n_1) \leq n_1-1$ and $G(n_2) = 2^h-(h+1) = n_2-h$. $G(n) = G(n_1)+G(n_2)+(h+1) \leq (n_1-1)+(n_2-h)+(h+1) = (n_1+n_2+1)-1 = n-1$.
□